8

~2019/06
William Mills
(Removed Confidential Marking 2021/04)

Devicetree Atom Table Format

Rationale

Devicetree and its binary format is is very string centric. While easy for development and easy to coordinate for deployment this has several disadvantages. Chief among these is size of the DTB files/images.

There are several issues with DTB size. Examples include:

- RTOS or bare-metal environments where code & data size are limited
- Early stage bootloaders like U-boot's SPL
- Universal U-boot images that work on a good number of boards
- Embedded/Tiny Linux that needs to fit in a 8MB SPI flash

Because of size limitations RTOS use cases are looking at code generation. A well defined best in class code generation pattern is valuable asset to the Devicetree framework. However, there are some use cases that could be more dynamic and yet still have size limitations. It is these use cases that would benefit the most from this proposal.

Idea

If the DTB and the OS have a common dictionary of strings with assigned index values, then neither one needs to store the actual values of the strings at runtime to do the basic operations needed for device tree.

Looking up a string value in the common dictionary will return an index and that index is referred to here as an atom. The same index is returned for all string inputs that compare equal to each other.

Thus:

Compare	To	Equality Operation
atom1	atom2	atom1 == atom2
atom1	string2	strcmp(atom_to_string(atom1), string2) == 0
string1	string2	strcmp(string1, string2) == 0

A new DTB format will be defined that can contain 0 or more atom table fragments. Each atom table fragment will contain a definition of the range of indexes it defines, a hash of the contents, and an optional offset to the atom table itself.

The new format will also define new structure elements that allow the replacement of inline strings with atom table indexes and replace string table offsets with atom table indexes.

Use of atom tables will not preclude use of a string table nor use of inline strings in structure elements.

Some useful combinations of these include:

- Bare metal RTOS with per-build DTBs
    - All or majority of DTB using atoms
    - No string table or Atom table needed in DTB
    - Inline string property values when needed
    - libfdt APIs uses atoms
    - No Atom table needed in OS
    - No Node string or property name , property value enum Strings needed in OS
    - True string property values still available
- DTB set with shared atom table
    - A set of DTBs & DTBOs all created at the same time
    - A single atom table is created with the string values of all the DTBs and DTBOs
    - This atom table is stored in the DTB set with full string values
    - The other DTBs of the set contain no node name, property name, or property value enum strings. True property value strings are still stored inline
    - Eliminates redundant strings that would have been stored in multiple DTBs
    - OS still uses libfdt APIs by strings
    - OS still has the string value for every atom in use so can compare atoms (DTB) to strings (libfdt APIs) and can pretty print the DTB structure
- DTB and OS with partial Atom tables
    - Mixed case of above two
    - DTB and OS have an agreed common dictionary that is stable and not stored
    - Still used atoms for all node names, property names, and property value enums
    - Atom table for stable dictionary is not stored in DTB and perhaps not in OS
    - Atom table for unstable parts stored in DTB

Purposed Details

DTB format changes

Strings in DTB today are encoded inline or stored in the DTB's string table. Node names and string property values are encoded inline in the DTB structure. Property names are encoded as a 32 bit offset into the DTB's string table. All strings are null byte terminated. (Properties may also be byte sequences. Such byte sequences are not necessarily null byte terminated and are considered out of scope for this proposal.

This proposal would add a few more sections to the DTB header and a few new tokens to the structure stream.

New header fields:

offset to atom fragment headers

size (or number) of atom fragment headers

Atom Fragment header

- starting index
- number of indexes
- offset to atom table or 0 if not present
- size of atom table (valid even when offset is 0 so HASH size is known)
- SHA256 HASH of atom table content
- pad to 8 bytes

Atom Table

- Array of N 32 bit offsets into the atom table
    - where N == number of indexes specified in header
    - An offset == 0 means the value of the string is not loaded
- Byte array of null byte terminated strings
- pad to 4 bytes (8?)

New Structure Tokens:

- FDT_BEGIN_NODE_ATOM
    - Node name atom (32 bit value)
    - [Node address cell sequence,
        - 1 cell for 32 bit values,
        - 2 cells for 64 bit values,
        - 0 cells if no unit address was given
    -]
- FDT_PROP_ATOM
    - Prop name atom (32 bit)
    - property value byte sequence same as FDT_PROP
        - no atom in this sequence, if used in source would expand to string inline

FDT_PROP_ATOM_VALUE_ATOM

Prop name atom (32 bit)

Prop value atom(s)

Normally a single atom but can be a sequence of atoms

Q: sequence of atoms implies comma seperation?

Each Atom in the DTB is in one of three states:

- Valid with no atom table present
- Valid with no string present in atom table (sparse atom table entry)
- Valid w/ string value

Partitioning of Atom Index Space

The value of this proposal increases if there is a well defined dictionary of standard strings. To this end this proposal suggests that the atom index space be partitioned into standardized atom space and ad-hoc atom space.

Since the atom space is a 32 bit value the values would be

- 0x0000_0000 to 0x7FFF_FFFF Standard atom space
- 0x8000_0000 to 0xFFFF_FFFF Ad-hoc atom space

It is suggested that once per year devicetree.org release a new standard atom table. New definitions are added to the table; existing definitions are never deleted or modified. Standard atom may be deprecated in new releases of the std atom table. Deprecation is a hint to DTB & OS image builders that the atom is a candidate for sparse atom table removal. It is up to the system builder to define the policy but an example policy could be: don't supply strings for any atom that has been deprecated for 5 years.

Devicetree.org needs to decide on a policy for what strings are candidates to go into the standardized space. There are arguments for being very lieral, for being very conservative, and for something in the middle.

The DTB structure allows the atom tables of the standard and ad-hoc spaces to be specified in fragments. It is expected that a fragment is supplied for each years portion of the standard atom space. Older versions may be more likely to be supplied w/o string definitions.

The ad-hoc index space is defined by the the OS builder. It can be ignored, used monolithically or used in multiple segments as the standard space.

Use of Atom's in OS Code

The use of atom id's in the OS code and libfdt APIs is entirely optional. The OS can choose to ignore the atom tables and just use strings.

OS strategies on atom usage can be:

- Include no atom tables of its own and use only strings
  Require that all DTBs come with complete atom tables
  Atom tables still provide value in
    - Reducing redundancy in a set of DTBs by unifying all strings into one base DTB with just the atom tables
    - Allowing small memory space OSes to consume the structure DTB w/o the atom definition DTBs while not having to deal with atom space management in the main OS
- Include only the standard atom space defined when the OS was first released
    - Allows DTBs to be smaller but still have a very simple model of atom space management
    - If full strings are included for the atom tables, the OS code can still use strings
- Use the atom table format features, any one or more of:
    - Define its own ad-hoc atom space
    - Leave out string definitions for some of the atom space
    - Use atom indexes in the OS's libdft APIs and OS data structures

Use of atom indexes in libfdt APIs and OS data structures

To use atom indexes in the OS code and data structures one could just use #defines but that would be very awkward. A strategy that allows strings to be used in the source code but atom ids be used in the object code is desirable.

Note: this process is similar to doing natural language translation in applications. Any technique used for that problem would be a candidate for string to atom mapping.

Dual value macro technique

One technique is to use a macro like

```
#ifdef USE_ATOMS

#define ATOM_STRING(atom, string)       atom

#else

#define ATOM_STRING(atom, string)       string

#endif
```

A code scanner would scan the source code and create and/or verify an atom to string table map while replacing any 0 atom values with newly allocated atom indexes. This scanning could be done as a preprocessor step once for each compilation or done statily to update the source code. A mixed strategy of putting standard space indexes into the source and creating the ad-hoc space on the fly would also make sense.

Compiler/Linker string constant folding technique

This technique relies on the compiler and linker process to fold all string constants of the same value into pointers to the same data location in a constant data table. A macro would be required to put all atom strings values into the same linker section.   The whole image would be prelinked and then another tool pass would construct the atom table from the collected strings. After this the atom table would be include in the whole program. Optionally the string table can be marked as a noload section.

Note: This technique requires that the toolchain coalesce/fold all strings in the section without any duplicates. A “best effort” folding that handles most usage but misses corner cases will not do. The tool that produces the string offset to atom index table should check and error out on duplicate string values.